Module: Mobile Application development (Android)

Session 51: Google maps, GPS, Camera Activity and File handling assignment Part 2 (Practice)

### Intro

### For this lab you will be developing a new GPS recording application called WalkAbout. The purpose of the application is to allow users to record their GPS location information as they travel. While the application records the user's GPS data, it displays it back to the user in the form of a path drawn on top of a Google Map. While recording data, the user can launch a Camera activity that will capture and store pictures on an SD-Card. When finished recording, the application gives the user the option of storing the current GPS data as a private application file to be loaded and displayed at a later time.

### Objectives

### At the end of this lab you will be expected to know:

### How to incorporate Google Maps into an application.

### How to register for and receive GPS location information.

### How to draw graphics on the screen using the Canvas class.

### How to create Google Maps Overlays.

### How to use the Camera.

### How to write data to the SD card.

### How to create and delete private application files.

### How to launch and receive results from Activities.

### 3.2 Retrieving GPS Location Information

### In this subsection, you will implement the functionality necessary to monitor and record GPS location information. You will monitor GPS location information by making the WalkAbout Activity register itself with the GPS LocationProvider. By doing this, the WalkAbout Activity will be notified by the GPS LocationProvider when the location changes. However, before the WalkAbout Activity class can register itself with the GPS LocationProvider, it must implement the com.google.android.maps.LocationListener interface. You will start by making the WalkAbout Activity implement the LocationListener interface.

### You will then implement a "Start/Stop" MenuItem that will toggle the WalkAbout Activity between actively-recording and recording-stopped states. While in the actively-recording state, the WalkAbout Activity will be registered to receive updates from the GPS LocationProvider. As the WalkAbout Activity receives notices about location changes, it will re-center its MapView about the new location. It will also store changes in location as latitude-longitude coordinates in an ArrayList. Additionally, while in the actively-recording state the "Start/Stop" MenuItem will display the word "Stop" to indicate that clicking on the MenuItem will cause the WalkAbout Activity to switch to the recording-stopped state.

### While in the recording-stopped state, the WalkAbout Activity will no longer be registered to receive updates from the GPS LocationProvider. Additionally, while in the actively-recording state the "Start/Stop" MenuItem will display the word "Start" to indicate that clicking on the MenuItem will cause the WalkAbout Activity to switch to the recording-stopped state. Note that when the Application starts up, it is by default in the recording-stopped state. When the user switches from the recording-stopped state to the actively-recording state, the ArrayList used to record changes in location should be cleared. See the Figure below for more details on the use case:

### [https://sites.google.com/site/androidcoursearchive/_/rsrc/1266438588854/labs/lab-5/StartStopMenuItem.png?height=147&width=400](https://sites.google.com/site/androidcoursearchive/labs/lab-5/StartStopMenuItem.png?attredirects=0)

### 3.2.1 Implement LocationListener Interface

### The WalkAbout Activity class will receive updates from the GPS Location Provider by registering itself with the Provider. In order to register with the Provider, the WalkAbout Activity must implement the LocationListener interface. This interface provides a set of callback methods that the Provider will call when the location changes, the Provider is enabled, the Provider is disabled, or when the status of the Provider changes.

### In particular, you will monitor and record location changes, and stop recording in the event that the Provider is disabled for some reason. The WalkAbout class records location changes in an ArrayList<GeoPoint> member variable named m\_arrPathPoints. It also maintains the current recording state (*whether it is currently recording or not*) in a boolean member variable named m\_bRecording. Perform the following initializations in the initLocationData() method.

### Initialize m\_arrPathPoints to a new ArrayList<GeoPoint>.

### Initialize m\_bRecording to false.

### Make the WalkAbout MapActivity class implement the LocationListener interface:

### There is a total of four methods that you must implement. See the Android Documentation on [LocationListener](http://developer.android.com/reference/android/location/LocationListener.html) for a complete listing and description of each.

### *Of the four methods, you only need to fill in two of them, onLocationChanged(...) and onProviderDisabled(...).*

### *You still need to declare the other two methods but you can leave them as empty stubs.*

### Record location changes in the onLocationChanged(...) method.

### *The Location argument passed in has both degrees latitude and longitude values, represented as type double. Some of the Google Maps classes that you will be using later on make use of the GeoPoint class, which represents latitude and longitude as int values of microdegrees. This is why you are storing an ArrayList of GeoPoints and not Locations. You will have to convert from Location to GeoPoint.*

### *1 degree is equal to 1E6 microdegrees. The WalkAbout Activity class has a constant for this value named GEO\_CONVERSION*

### *99.123456 degrees = 99123456 microdegrees*

### Get the degrees latitude value from the Location argument by calling the Location.getLatitude() method and convert it to microdegrees.

### Convert by multiplying the returned double by the WalkAbout.GEO\_CONVERSION integer constant.

### Get the degrees longitude value from the Location argument by calling the Location.getLongitude() method and convert it to microdegrees.

### Instantiate a new GeoPoint object, passing into the constructor the latitude and longitude values you just converted into microdegrees.

### You will have to cast these to type int as they are currently of type double.

### Add the GeoPoint to m\_arrPathPoints.

### In the onProviderDisabled(...) method, instruct the WalkAbout Activity to stop recording location changes.

### Do this by making a call to WalkAbout.setRecordingState(false). You will implement this method in the sections that follow.

### 

### 3.2.2 Dynamically Update the "Start/Stop" MenuItem Text

### When the Options Menu is displayed, the text of the "Start/Stop" MenuItem should be set to the R.string.startRecording if the activity is not recording and R.string.stopRecording if the activity is recording.

### 3.2.3 Implement "Start/Stop" MenuItem

### 

### When the "Start/Stop" MenuItem is clicked, the recording state should be toggled. If the activity is currently recording, then the activity should stop recording. Conversely, if the activity is not recording, then the activity should start recording.

### The functionality for setting the recording state will be encapsulated in the WalkAbout.setRecordingState(boolean recording) method. Passing in a value of true indicates that the activity should start recording and passing in a value of false indicates that the activity should stop recording.

### Set and fill in the "Start/Stop" OnMenuItemClickListener so that it toggles the recording state by calling WalkAbout.setRecordingState(...) with the proper value.

### This is all this method should do.

### Make use of the m\_bRecordingState member variable to determine the current recording state.

### Fill in the setRecordingState(...) method:

### You should update the m\_bRecording member variable with the new state.

### If the new state is not recording you should tell the GPS Provider to stop sending you Location Updates.

### You can do this by calling the LocationManager.removeUpdates(...) method and pass in the LocationListener that you want to un-register.

### Otherwise, if the new state is recording, you should re-initialize your recording data.

### All previously recorded GeoPoints should be erased. Do not instantiate a new list of GeoPoints, just clear your current one.

### *You should get used to trying to re-use objects when possible. It is expensive to create new ones and it may be some time before the old ones get garbage collected.*

### Initialize your list of GeoPoints with the last known location from the GPS Provider.

### You can do this by calling the [LocationManager.getLastKnownLocation(...)](http://developer.android.com/reference/android/location/LocationManager.html" \l "getLastKnownLocation%28java.lang.String%29) method.

### You can then manually call your WalkAbout.onLocationChanged(...) method with the Location you just retrieved. This will update your list of GeoPoints for you.

### Tell the GPS Provider to start sending you Location Updates.

### You can do this by calling the LocationManager.requestLocationUpdates(...) method.

### Pass in the String name of the Location Provider you want to receive Location Updates from

### Pass in WalkAbout.MIN\_TIME\_CHANGE, to specifiy the minimum amount of time between updates.

### Pass in WalkAbout.MIN\_DISTANCE\_CHANGE, to specifiy the minimum amount of distance between updates.

### Pass in the LocationListener which should receive the updates.

### If you run your application, it should now record GPS Location changes. You can test this by adding a Toast notification to the WalkAbout.onLocationChanged(...) method that prints out the new location. Don't forget to remove the Toast after you've finished testing.

### If you are running this on the emulator, you can simulate location changes on the emulator two different ways. You can do this from the DDMS perspective in Eclipse, or from the Console. For instructions on how to open up a console, see the Android reference on [Using the Console](http://developer.android.com/guide/developing/tools/emulator.html#console).

### DDMS:  see the DDMS documentation on [Emulator Controls](http://developer.android.com/guide/developing/tools/ddms.html#emulator-control). Feel free to use the testGPS.gpx file included in the root folder of the skeleton project as a test path.

### Console: see the documenation on Location Services

### 

### 3.2.4 Center MapView on Current Location

### As the application stands now, when you change location, the MapView does not move. It is therefore possible for your MyLocation beacon to move off of the MapView. You will now change this so that when your position changes, the MapView will center itself about the MyLocation beacon. This is done using the MapView's MapController object. The MapController can be used for a number of different things including changing the zoom level.

### In the WalkAbout.onLocationChanged(...) method, retrieve the MapController from your MapView by calling the MapView.getController() method.

### Use the MapController.animateTo(Point point) method to force the map to scroll so that the current location is now in the center.

### You should now be able to run your application to ensure that the Map is always centered about the device's current location.

### 

### 3.3 Path Overlay

### For this subsection, you will implement your own Overlay class that draws a path on top of a MapView. You will begin by filling in the PathOverlay class so that it draws a green starting circle at the first point in the path, and then draws a red line to connect each subsequent point in the path. The PathOverlay class will contain a reference to a List of GeoPoints. You will then add the PathOverlay to the WalkAbout Activity's MapView object, setting it up so that the PathOverlay uses the WalkAbout Activity's recorded list of GeoPoints as the path that it should draw.

### [https://sites.google.com/site/androidcoursearchive/_/rsrc/1266438624028/labs/lab-5/MyLocationOverlay.png?height=400&width=266](https://sites.google.com/site/androidcoursearchive/labs/lab-5/MyLocationOverlay.png?attredirects=0)

### 3.3.1 Fill in the PathOverlay Class

### Begin by filling in the constructor:

### Set m\_arrPathPoints to the pathPoints constructor argument. This is the list of GeoPoints that represents the path that this overlay class will draw. Notice that it is declared as final; from the perspective of the PathOverlay class this object is immutable/Read-Only. You should not attempt to modify the contents of this List, you should only read from it.

### Initialize each of the other four member variables with new default constructed objects. You will use these objects to help you draw the path. Instead of recreating them every time the draw method is called, you will simply re-use them to improve performance.

### The only thing that this object is supposed to do is draw the path overlay, and thus it has only one method, draw(...). The PathOverlay should draw the first point as a green circle and connect each subsequent point with red a line. If the list of GeoPoints is empty or null, the method should do nothing.

### The Canvas argument that is passed into the draw(...) method is what performs the drawing for you. You can get something to show up on the screen by asking the Canvas object to draw something for you. You do that by telling it what to draw, where to draw it, and what color it should use. The MapView argument is a reference to the Map on which this overlay will be drawn. Don't worry about the shadow argument for now.

### Start by drawing the starting circle:

### Set the color to green by calling the Paint.setARGB(...) method on m\_paint, passing in the Red, Green, Blue, and Alpha values.

### Specify where to draw the circle by calling the RectF.set(...) method on m\_rect. This outlines a Rectangle in which to draw the circle. The RectF.set(...) method takes in the x values of the left & right edges and the y values of the top & bottom edges.

### Before you can call the RectF.set(...) method, you need to translate your first GeoPoint coordinate into screen coordinates.

### Call the MapView.getProjection() method to retrieve a projection object that will perform the translation for you.

### You can now call the Projection.toPixels(...) method, passing in your first GeoPoint and the m\_point member variable.

### After the call, m\_point will contain the x-y screen coordinates.

### Use the following line to set the bounding rectangle:

### m\_rect.set(m\_point.x-START\_RADIUS, m\_point.y-START\_RADIUS, m\_point.x+START\_RADIUS, m\_point.y+START\_RADIUS);

### Tell the canvas to draw the circle by calling Canvas.drawOval(...), passing in your bounding rectangle and the color it should use.

### You should now start drawing your path. For each point in the list of GeoPoints, draw a red line starting from the current point to the point following it. For example, if your list has 3 points in it, you should draw a line connecting point-1 to point-2 then you should draw a line connecting point-2 to point-3.

### Use m\_point2 to store the second set of screen coordinates.

### The line should have a thickness equal to PathOverlay.PATH\_WIDTH. You can set the width of the line by calling the Paint.setStrokeWidth(...) method.

### You can use the Canvas.drawLine(...) method to draw the line.

### *Note: You won't use the m\_rect to draw the path, you only use it to draw the starting circle.*

### 

### 3.3.2 Add a PathOverlay to Your Map

### In the WalkAbout activity class

### Initialize and add a PathOverlay to your MapView in the WalkAbout.initLayout() method.

### Since the PathOverlay only gets redrawn when the MapView gets redrawn, you need to notify the MapView to redraw itself any time you update your path/recorded-list-of-GeoPoints.

### You tell the MapView to redraw itself by calling MapView.postInvalidate();

### When you add GeoPoints to your recording-list/path, you need to redraw the MapView.

### When you clear your recording-list/path, you need to redraw the MapView

### You should now be able to run your application to ensure that when recording, your path gets continuously updated and displayed on the MapView.

### 

### 4. Using The Camera

### 4.1 Camera Preview

### Android allows applications direct access to the Camera Hardware. In the section that follows you will implement the WalkAbout Activity "Take Picture" MenuItem. The MenuItem should only be enabled while the Application is in the actively-recording state. When the MenuItem is clicked, it will launch the CameraPreview Activity. It is your job to extend this Activity so that it can capture a picture and save it to the SD-Card. The CameraPreview Activity was borrowed from the Google API Demos application, and currently is only able to display a full screen preview of what the camera is looking at.

### The CameraPreview Activity will take a picture once the user touches anywhere on the screen. Once the picture is taken, it will be saved on the SD-Card and the file-name will be returned to the WalkAbout Activity as a result. When the user returns to the WalkAbout Activity, a short Toast notification containing a success message and the file-name the picture was saved to will be displayed to the user. Alternatively, the user can hit the back button without taking a picture to return to the WalkAbout Activity, effectively canceling the CameraPreview Activity. Should the CameraPreview Activity be canceled, the user will see a short Toast notification containing a failure message.

### [https://sites.google.com/site/androidcoursearchive/_/rsrc/1266438789324/labs/lab-5/TakePicture2.png?height=147&width=400](https://sites.google.com/site/androidcoursearchive/labs/lab-5/TakePicture2.png?attredirects=0)

### 4.1.1 Dynamically Update Options Menu

### When the WalkAbout Options Menu is displayed, the "Take Picture" MenuItem should only be enabled if the Activity is recording GPS location changes. If the Activity is not recording, the MenuItem should be disabled.

### 

### 4.1.2 Launch the CameraPreview Activity

### When the "Take Picture" MenuItem is clicked, the CameraPreview Activity should be launched. Currently, the CameraPreview Activity class does not have the ability to take a picture, but rather only provides a preview image. You will add the functionality to this class to take the picture in the next section. Before you can do that, you need to update your the AndroidManifest.xml:

### Add the following <uses-permission ... /> and <uses-feature ... /> lines to your AndroidManifest.xml file. These lines should be nested inside of the <manifest></manifest> tags:

### <manifest ...>     <uses-feature android:name="android.hardware.camera" />     <uses-permission android:name="android.permission.CAMERA"/> </manifest>

### 

### In the WalkAbout Activity, set and fill in the "Take Picture" OnMenuItemClickListener so that it launches the CameraPreview Activity and recieves a result back from the activity.

### Create a new Intent, passing in a reference to this application Context and the edu.calpoly.android.walkabout.CameraPreview Class object.

### Call Activity.startActivityForResult(...), passing in the Intent object and the WalkAbout.PICTURE\_REQUEST\_CODE integer constant.

### 

### 4.1.3 Take a Picture from the CameraPreview Activity

### The CameraPreview Activity class comes from the Google Android API Demos application which provides sample applications for a number of different Android API's. In order to display a Camera Preview you need to use some advanced Android features. Instead of having you implement this yourself, it's provided for you so that you can see how its done.

### Notice the inner Preview class that extends the SurfaceView class. This is the View that renders the Camera Preview Image and displays it to the user.

### The SurfaceView class is used when rendering is too computationally expensive to be done from the GUI thread or the View needs to be refreshed very quickly.

### The Preview class uses a Camera object and its Camera.Parameters inner class to retrieve the camera image data and manipulate its settings.

### A single getter method was added to the Preview class for the purposes of retrieving a reference to the Camera object. This was done so that you can add functionality to the CameraPreview Activity class to take a picture. When a user clicks anywhere on the CameraPreview's mPreview object (which occupies the entire screen) the Preview object's camera should take a picture.

### Set an OnClickListener for the CameraPreview's Preview object and fill in its onClick(...) method

### Retrieve the Preview object's Camera and call its takePicture(...) method.

### This method takes 3 separate listeners, each of which implement a different callback that will be called during different stages of the photo taking process. All of these can be null if we don't want to respond to any of these stages.

### The First is a ShutterCallback that will be called once the image has been captured.

### The Second is a PictureCallback that will be called once the raw image data has been generated.

### The Third is another PictureCallback that will be called once the jpeg data has been generated. This is the only one we are interested in.

### Pass in null for the first two parameters and pass in a reference to *this* CameraPreview instance for the third. CameraPreview implements the PictureCallback interface. This will make CameraPreview's onPictureTaken(...) method be called once the jpeg data has been generated.

### You should be able to run your application and verify that you can launch the CameraPreview activity. You should be able to see a preview of what the camera is seeing. If you add a Toast notification to the CameraPreview.onPictureTaken(...) method, you should see the notification when you click anywhere on the screen.

### 

### 4.1.4 Save the Picture to SD-Card

### Now that the picture has been taken, we need to save the jpeg data into a file on the SD-Card when the CameraPreview.onPictureTake(...) method is called. In order for your application to write data to the SD-Card, it must declare that it uses the WRITE\_EXTERNAL\_STORAGE permission in its manifest.

### Do so by adding the following <uses-permission ... /> line to your AndroidManifest.xml file. This line should be nested inside of the <manifest></manifest> tags:

### <manifest ...>     <uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"/> </manifest>

### Now you can implement the File-Output functionality by filling in the CameraPreview.onPictureTaken(...) method:

### Instantiate a new java.io.File object, passing in the path to the SD-Card directory and the name of the File you want to create.

### You can retrieve the path to the SD-Card directory by calling the static Environment.getExternalStorageDirectory() method.

### To guarantee that all of your file names are unique, use the Current System Time appended with the ".jpg" extension as your file name.

### *For example: If you call System.currentTimeMillis() your file will look something like "1111111111111.jpg"*

### Instantiate a new java.io.FileOutputStream object passing your new File object into the constructor.

### Write the byte[] data argument passed into the onPictureTaken(...) method out to the FileOutputStream.

### Flush the stream and close it.

### You must catch any IOExceptions that occur.

### Should an IOException occur, set the result of the Activity by calling the Activity.setResult(...) method.

### Pass in Activity.RESULT\_CANCELED as the result code and null as the Intent data to indicate that the Picture was not taken successfully.

### call the Activity.finish() method

### If the file was written successfully:

### Default instantiate a new Intent object. You can use this Intent object to pass the path to the image file back to the WalkAbout Activity.

### Use the Intent.putExtra(...) method to store the absolute path to the image file. You can use CameraPreview.IMAGE\_FILE\_NAME as the key.

### Set the result for the Activity with Activity.RESULT\_OK and the Intent data, then finish the Activity.

### 

### 4.1.5 Handle the Result

### After the CameraPreview Activity finishes, the WalkAbout Activity needs to handle the result of the Intent. The WalkAbout Activity should make a Toast Notification displaying either success or failure as returned by the CameraPreview Activity. If the pictures was taken successfully, the path to the created image file should be displayed in the notification. Do this by overriding and filling in the Activity.onActivityResult(...) method:

### Make sure to call super.onActivityResult(...).

### Test the requestCode argument to ensure that it matches the request code that you used when you launched the CameraPreview activity.

### If the request codes don't match, then don't do anything and just return. If they do match then continue on.

### Test the resultCode argument.

### If it equals Activity.RESULT\_CANCELED then print a Toast Notification displaying the R.string.pictureFail string resource.

### If it equals Activity.RESULT\_OK then print a Toast Notification displaying the R.string.pictureSuccess string resource and the full image file path.

### You can retrieve the image file path by calling getExtras().getString(CameraPreview.IMAGE\_FILE\_NAME) on the Intent data argument.

### You should be able to run your application and verify that the Toast Notification displays the proper text when the CameraPreview Activity finishes.

### 

### 

### 5. Private Application Files

### Android allows you to Create, Write, Read and Delete local files. These files become private to the application that created them by default. You have the option of overriding this privacy mechanism, allowing them to be shared with other applications.

### 5.1 Creating, Writing, & Deleting Files

### You will now implement WalkAbout Activity's "Save" MenuItem. This MenuItem's OnMenuItemClickListener should only make a single call to the WalkAbout.saveRecording() method. The functionality necessary to save the current recorded path should be composed in the WalkAbout.saveRecording() method. You should fill in the saveRecording method so that it properly writes out only the contents of m\_arrPathPoints to a private application file. You should only ever create one file and it should be truncated/cleared each time you write to it.

### [https://sites.google.com/site/androidcoursearchive/_/rsrc/1266438828245/labs/lab-5/SaveMenuItem.png?height=297&width=400](https://sites.google.com/site/androidcoursearchive/labs/lab-5/SaveMenuItem.png?attredirects=0)

### The format of the file is quite simple. The contents of m\_arrPathPoints will be written into a single continuous line. The latitude and longitude of each GeoPoint in m\_arrPathPoints should be written out in that order, separated by a comma,andno spaces between them. Each GeoPoint should be separated by a semi-colon, and no spaces between them. So for example:

### Given:

### m\_arrPathPoints = [(lat1,long1),(lat2,long2),(lat3,long3)]

### The file should look like:

### lat1,long1;lat2,long2;lat3,long3;

### 

### If the Save was performed successfully, then a Toast notification should be displayed containing the R.string.saveSuccess resource string. If an exception is thrown you should display a Toast notification containing the R.string.saveFailed resource string. If there is no data to save, as is the case on the initial loading of the application, then a Toast notification should be displayed containing the R.string.saveNoData resource string. You are tasked to do this on your own, however, you should make use of the following hints:

### Use the R.string.geoPathFileName string resource as the filename

### You will have to make use of the [Context.openFileOutput(String name, int mode)](http://developer.android.com/reference/android/content/Context.html" \l "openFileOutput%28java.lang.String,%20int%29) and [Context.deleteFile(String name)](http://developer.android.com/reference/android/content/Context.html" \l "deleteFile%28java.lang.String%29)methods.

### You will have to use the Context.MODE\_PRIVATE constant.

### Documentation on the [java.util.Scanner](http://www.google.com/url?q=http%3A%2F%2Fjava.sun.com%2Fj2se%2F1.4.2%2Fdocs%2Fapi%2Fjava%2Fio%2FPrintWriter.html&sa=D&sntz=1&usg=AFrqEze_BnV_mCMubO1dK8AjHv7i4mKx4Q) class *(This is merely a suggestion, you are free to use whatever Java I/O classes you like to write to the file)*.

### 5.2 Reading Files

### Your last task is to implement the WalkAbout Activity's "Load" MenuItem. This MenuItem's OnMenuItemClickListener should only set the recording state to false and make a call to the WalkAbout.loadRecording() method. The rest of the File Loading functionality should be implemented in the loadRecording() method. You should fill in the loadRecording method so that it properly initializes m\_arrPathPoints to contain only the data in the file that saveRecording() writes out. Once finished, the path loaded from the file should be displayed exactly as it was when it was first recorded.

### [https://sites.google.com/site/androidcoursearchive/_/rsrc/1266438857724/labs/lab-5/LoadMenuItem.png?height=294&width=400](https://sites.google.com/site/androidcoursearchive/labs/lab-5/LoadMenuItem.png?attredirects=0)

### If the Load was performed successfully, then a Toast notification should be displayed containing the R.string.loadSuccess resource string. If an exception is thrown, you should display a Toast notification containing the R.string.loadFailed resource string. You are to do this task on your own, however, you should make use of the following hints:

### You will have to make use of the [Context.openFileInput(String name) method](http://developer.android.com/reference/android/content/Context.html" \l "openFileInput%28java.lang.String%29) in order to open the file.

### Documentation on the [java.util.Scanner](http://www.google.com/url?q=http%3A%2F%2Fjava.sun.com%2Fj2se%2F1.5.0%2Fdocs%2Fapi%2Fjava%2Futil%2FScanner.html&sa=D&sntz=1&usg=AFrqEzfaXJBBhubxcWolTQfaPxxxvFLwGg) class *(This is merely a suggestion, you are free to use whatever Java I/O classes you like to read from the file).*